Semestrálna práca MI-PAP 2013/2014:

Násobenie matíc

Martin Klepáč

Pavol Kušlita

May 3, 2014

1 Definícia problému

Našou úlohou bolo vytvoriť program, ktorý implementuje násobenie matíc formou ako klasického algoritmu, tak aj s použitím Strassenovho algoritmu.

2 Formát vstupu, výstupu

Formálne, vstup vyjadríme pomocou

- *A, B* = vstupné matice
- *Ax, Ay, Bx, By* = dimenzie vstupných matíc A, B

Výstupom algoritmu je matica *C* s dimenziami *Ax*, *By* za podmienky, že *Ay* = *Bx*. V opačnom prípade nie je možno vykonať násobenie vstupných matíc.

3 Implementácia sekvenčného riešenia

Sekvenčný algoritmu pozostáva z trojice cyklov, ktoré prechádzajú matice *A*, *B* v nasledovnom poradí:

1. riadok matice *A*
2. stĺpec matice *B*
3. stĺpec matice *A*, ktorý zároveň predstavuje riadok matice *B*

Vo najvnútornejšom cykle sa potom vykoná samotný výpočet popísaný pseudokódom nižšie:

```
C[i][j] += A[i][k] * B[k][j];
```

Zložitosť takéhoto výpočtu je potom intuitívne $O(n^3)$. Zároveň tento triviálny algoritmus nevyužíva možnosti cache pamäte.

Vylepšenie tohto triviálneho algoritmu spočíva v použití techniky loop-tiling. Výsledkom je zdvojnásobenie celkového počtu cyklov na 6, pričom miesto sekvenčného posunu indexov *i* až *k* posúvame indexy v našom prípade o 100, aby sme následne iterovali pomocnými indexami *ii*, *jj* respktíve *kk* medzi 0-99, 100-199 a tak ďalej. Výsledkom je rovnaké množstvo vykonanej práce pri efektívnejšom využití cache pamäte. Pseudokód popisujúci techniku loop tiling-u:

```
for i in 0..Ax
 for j in 0..By
  for k in 0..Ay
   for ii in i..i+100
    for jj in j..j+100
     for kk in k..k+100
      C[ii][jj] += A[ii][kk] * B[kk][jj];
```

Strassenov algoritmus je asymptoticky rýchlejší než štandardný multiplikatívny algoritmus. Jeho asymptotická zložitosť je $O(n^{2.8074})$. Jedna sa o rekurzivný algoritmus, ktorý maticu *A* a *B* rozdeli na podmatice *A11*, *A12*, *A21*, *A22* a podobne aj maticu *B*.

$A = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$

Následne do pomocných matíc *p1* až *p7* uloží čiastočné výsledky násobenia. V tomto kroku dochádza k zrýchleniu voči klasickému násobeniu, kedže dochádza iba k siedmim násobeniam v porovnaní s ôsmimi násobeniami v prípade klasického alogritmu. Pri týchto siedmich násobeniach opäť využívame Strassenov algoritmus až do určitej hranice, pri ktorej je už efektívnejšie používať klasický spôsob. Táto hranica závisí od implementácie a od hardvéru. V našom prípade bola empiricky stanovená na hodnotu 64. Na konci algoritmu sa medzivýsledky v pomocných maticiach *p1* až *p7* spoja a vytvoria matice *C1*, *C2*, *C3*, *C4*, ktoré spolu vytvárajú výslednú maticu C.

4 Implementácia paralelného riešenia pomocou OpenMP

Prostredie OpenMP sa ukázalo byť veľmi jednoduché na implementáciu triviálneho algoritmu s použitím viacerých vlákien. Vzhľadom na to, že objem práce v najvnútornejšom cykle je rovnaký pre všetky vlákna, použili sme statické rozdelenie záťaže priamo v prostredí OpenMP definované kľúčovým slovom *static*. Aby sme v prípade menších vstupných matíc v kombinácii s väčším počtom vlákien zamedzili plytvaniu prostriedkov, miesto jedného cyklu paralelizujeme dvojicu vonkajších cyklov s pomocou kľúčového slova *collapse*.

Kód definujúci paralelizáciu ako triviálneho algoritmu, tak aj algoritmu s použitím loop tiling-u vyzerá nasledovne:

```
#pragma omp parallel for collapse (2) default(shared) private(i,j,k)
schedule(static)
```

	1024x1024			2048x2048			4096x4096		
	CL	**LT**	**ST**	**CL**	**LT**	**ST**	**CL**	**LT**	**ST**
-n 1	11.33	10.02	4.85	125.20	79.60	34.15	1347.2	651.67	239.81
-n 2	5.84	5.47	2.82	56.44	41.22	19.82	565.86	327.14	138.82
-n 4	2.93	2.82	1.49	30.03	20.79	10.21	302.63	163.87	70.66
-n 6	1.96	1.94	1.10	20.59	13.96	7.35	203.99	109.19	52.04
-n 8	1.48	1.46	0.85	16.33	10.57	5.80	158.66	81.65	39.93
-n 12	0.98	1.07	0.60	10.74	7.05	3.96	114.60	56.16	27.91
-n 24	0.92	0.81	0.50	8.40	5.87	3.14	83.37	50.18	19.71

Table 1: Trvanie behu v sekundách na OpenMP (CL = classic, LT = loop-tiling, ST = Strassen)

Pri paralelizácií Strassenovho algoritmu sme využili koštrukciu *Task*. Jedná sa o základnú jednotku paralelizácie. Paralelizácia spočíva v tom, že násobenie medzivýsledkov *p1* až *p7* sme rovnomerne rozdelili do 4 skupin. Prvé 3 skupiny sa rozdelia do 3 taskov a poslednú skupinu bude násobiť riadiace vlákno. Na konci tejto kritickej sekcie sa použije konštrukcia *taskwait*, ktorá zablokuje riadiace vlákno, kým jednotlivé tasky nebudú dokončené. Záverečné spájanie medzivýsledkov do výslednej matice C sme taktiž rozdelili do 4 taskov.

5 Merania na CPU

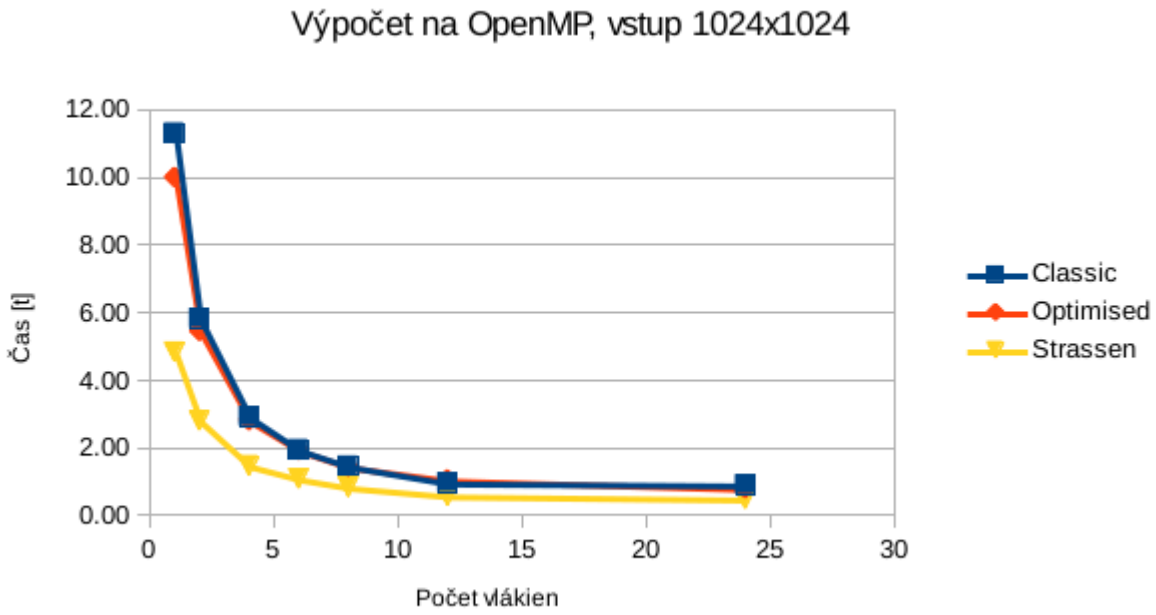
Pre potreby merania na serveri star.fit.cvut.cz sme vygenerovali trojicu pseudonáhodných štvorcových matíc o veľkosti 1024, 2048 a 4096. Program samozrejme dokáže spracovať aj iné ako štvorcové matice, ale pre férové porovnanie Strassenovho algoritmu s ostatnými riešeniami, používame práve štvorcové matice o veľkosti 2^n - naša implementácia Strassena automaticky dopĺňa vstupné matice na najbližšiu vyššiu takúto veľkosť.

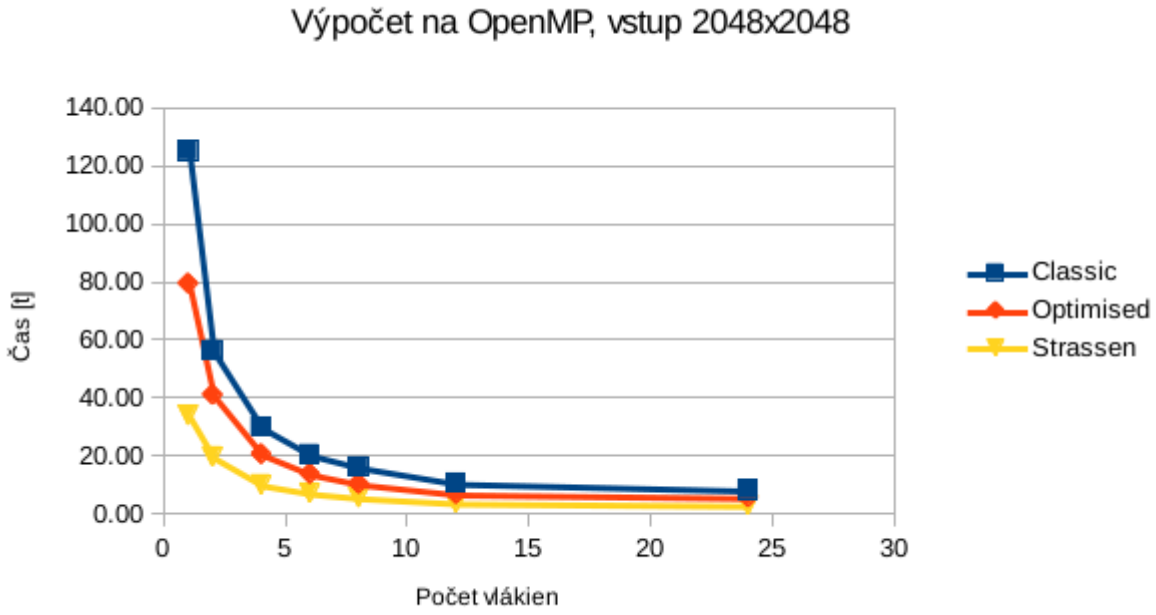
Namerané hodnoty sú uvedené v tabuľke 1 pre dané vstupné matice, počet vlákien 1, 2, 4, 6, 8, 12, 24 pre všetky hore popísané implementácie. Výsledné hodnoty predstavujú aritmetický priemer trojice meraní.

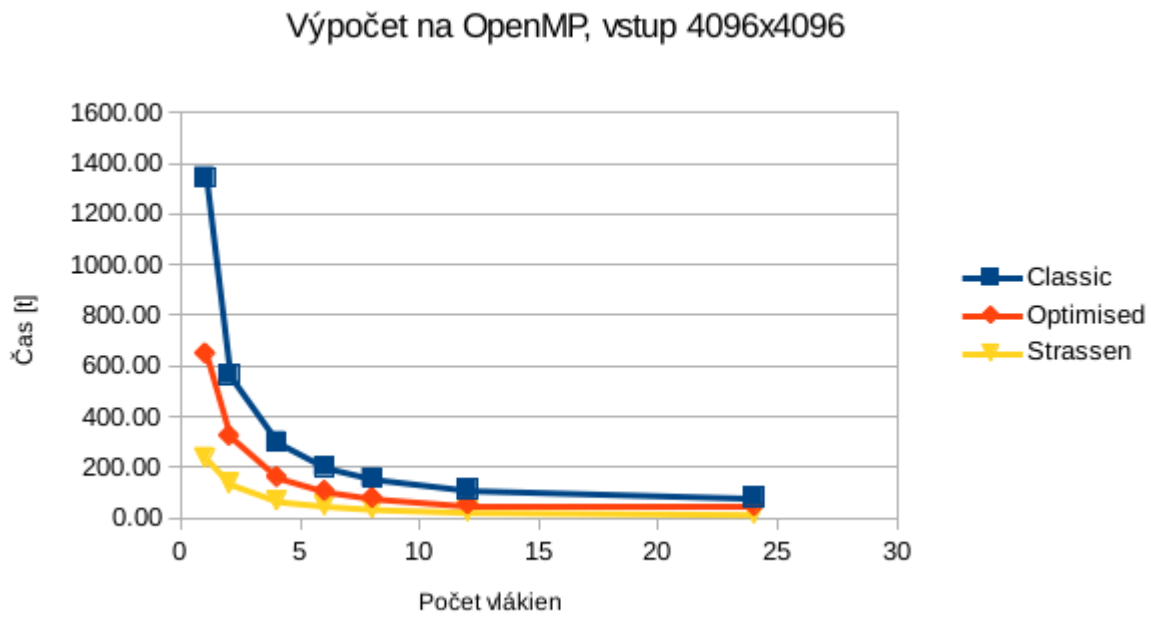
Nasledovné grafy znázorňujú ako paralelný čas, tak aj paralelnú cenu pre všetky tri vstupy pre variabilný počet vlákien.

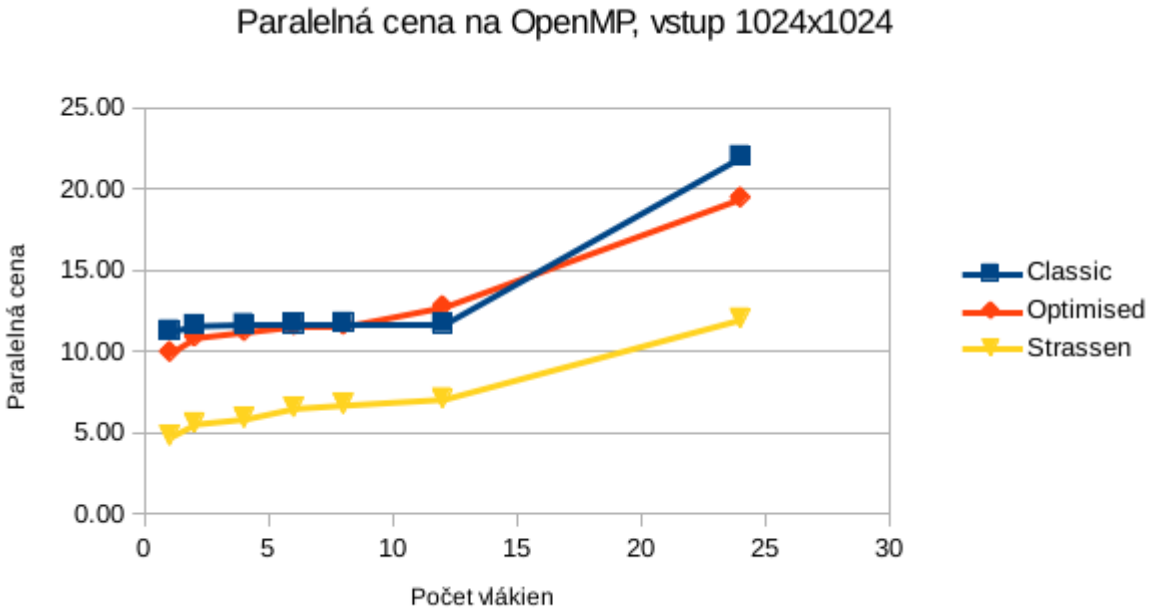
Ako môžeme vidieť, pre takmer všetky vlákna (výnimkou je 24 vlákien) a všetky implementácie sme dosiahli približne lineárne zrýchlenie, a teda konštantnú paralelnú cenu. Zvýšenie paralelnej ceny pre 24 vlákien spočíva v tom, že nami použitý hardvér nemal k dispozícii 24 fyzických procesorov, a teda zákonite došlo k prepínaniu kontextu v rámci jedného fyzického procesora. V porovnaní s predmetom BI-PPR, na ktorom sme pracovali s distribuovanou pamäťou, nedošlo k superlineárnemu zrýchleniu.

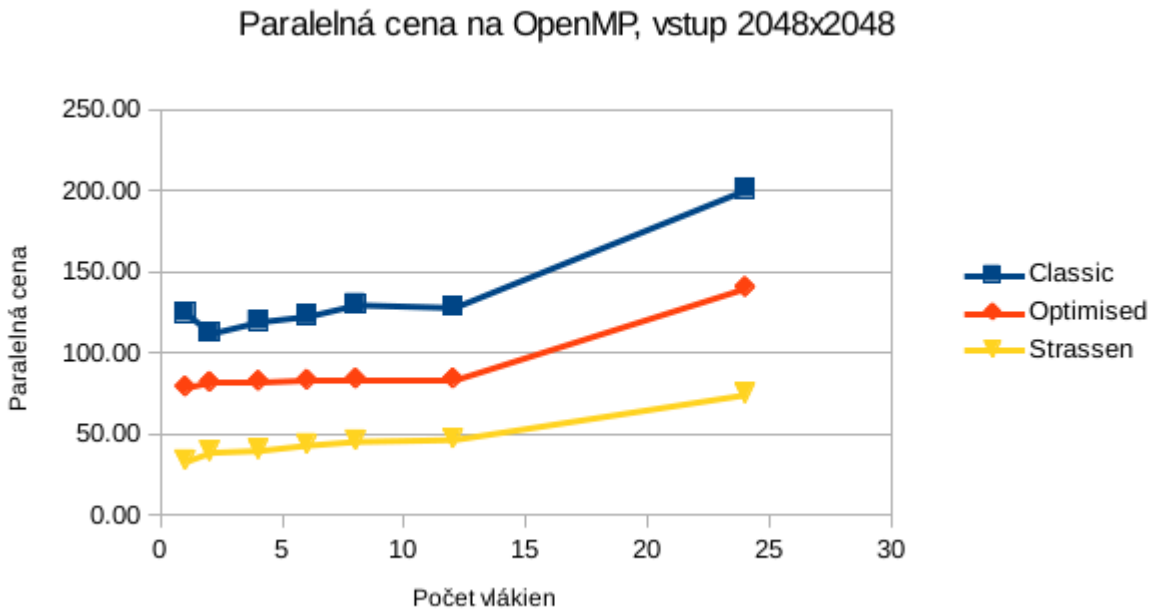
Ak by sme porovnávali jednotlivé implementácie medzi sebou, klasický a optimalizovaný algoritmus majú rovnakú asymptotickú zložitosť. Napriek tomu dosahuje optimalizovaná verzia zrýchlenie približne v rozsahu 10 až 50 percent v závislosti

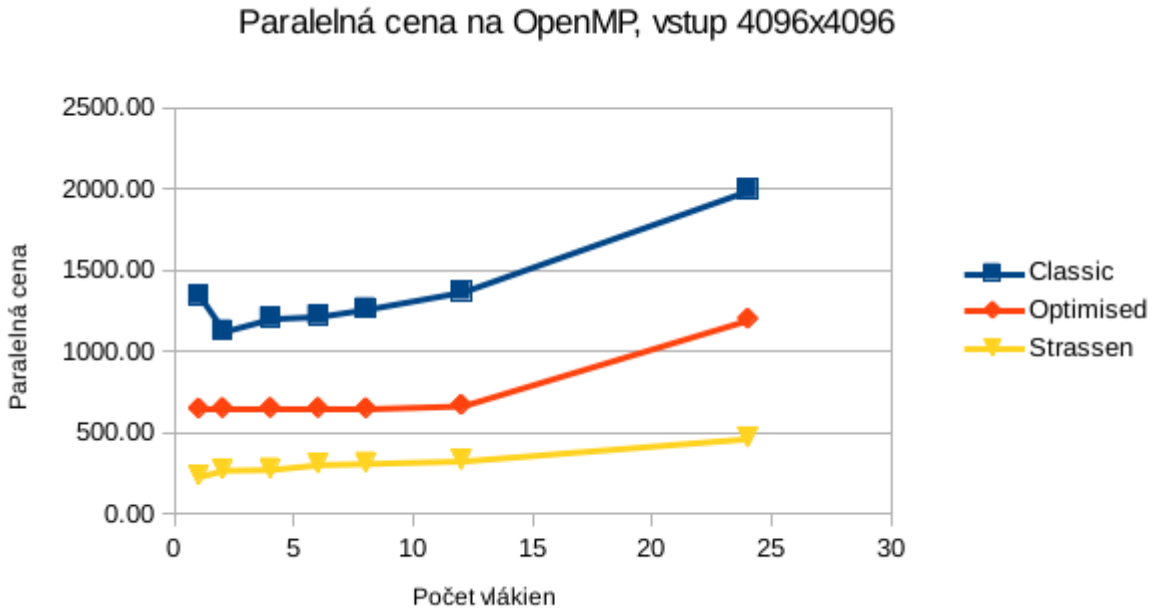

Výpočet na OpenMP, vstup 1024x1024
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Čas [t]
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen


Výpočet na OpenMP, vstup 2048x2048
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
Čas [t]
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen


Výpočet na OpenMP, vstup 4096x4096
1600.00
1400.00
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00
Čas [t]
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen


Paralelná cena na OpenMP, vstup 1024x1024
25.00
20.00
15.00
10.00
5.00
0.00
Paralelná cena
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen


Paralelná cena na OpenMP, vstup 2048x2048
250.00
200.00
150.00
100.00
50.00
0.00
Paralelná cena
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen


Paralelná cena na OpenMP, vstup 4096x4096
2500.00
2000.00
1500.00
1000.00
500.00
0.00
Paralelná cena
0
5
10
15
20
25
30
Počet vlákien
Classic
Optimised
Strassen

od veľkosti vstupu (čím väčší vstup, tým väčšie zrýchlenie). Strassenov algoritmus, ktorého zložitosť je $O(n^{2.8074})$, dosahuje v porovnaní s optimalizovanou variantou zrýchlenie približne o 50 percent.

6 Implementácia na GPU

Implementáciu na GPU sme vykonali v 2 krokoch. V prvom kroku sme sa sústredili na vytvorenie funkčného riešenia, ktoré pracovalo výlučne s *globálnou GPU pamäťou*. V druhom kroku sme sa časté prístupy do pamäte tohto typu snažili zminimalizovať pomocou *zdieľanej pamäte*, ktorá je súčasťou každého symetrického multiprocesora, a teda bližšie ku koncovým vláknam.

V prvej implementácii sme spúšťali kernel s parametrami *(matrixSize, matrixSize)*, čiže pre každý element výstupnej matice sme spustili 1 vlákno, ktoré v rámci kernelu vykonalo skalárny súčin i-teho riadku matice A s j-tym stĺpcom matice B a výsledok zapísalo do matice C na pozíciu [i][j]. Takto spúšťané riešenie je zároveň limitované hodnotou maximálneho počtu vlákien v rámci jedného bloku, ktorá sa líši v závislosti od Cuda Compute Capabilities (CC). Tento problém sa dá obísť zvýšením počtu blokov a rozdelením do viacerých dimenzií a naopak zmenšením počtu spúšťaných vlákien v 1 bloku tak, aby sa celkový počet spustených vlákien rovnal pôvodnej hodnote druhej mocnine *matrixSize*.

V druhej implmenetácii sme pristúpili k redukcii počtu prístupov do globálnej GPU pamäte. Miesto nej využijeme zdieľanú pamäť - jej problémom je veľkosť rádovo v desiatkach kB, a preto rozdelíme problém násobenia dvoch veľkých matíc na niekoľko násobení menších podmatíc, ktoré sa svojou veľkosťou zmestia do shared memory. Aplikujeme teda podobný princíp ako v prípade optimalizovaného sekvenčného riešenia, v ktorom sme využili loop tiling pre zvýšenie využitia cache pamäte.

Otázkou bolo preto určiť veľkosť submatice. Čím väčšiu veľkosť TILE_WIDTH sme volili, tým viac sme využívali shared memory, až sme došli k jej limitom. Tie sa zastavili na hodnote 64, pri ktorej program už nefungoval. Na druhej strane, s narastajúcou veľkosťou TILE_WIDTH klesá počet aktívnych blokov, ktoré sa o shared memory v rámci jedného symetrického multiprocesoru navzájom delia. Na základe empirického skúmania sme nakoniec zvolili TILE_WIDTH rovnú 16.

7 Merania na GPU

Merania sme realizovali na všetkých možných grafických procesoroch na školskom klastri. Výstupom je priemerný čas z trojice meraní pre vstupné matice o veľkosti 256, 512, 1024, 2048, 4096 a 8192. Testy sme realizovali na grafických procesoroch NVIDIA GTX 470, GTX 480, GTX 590 a Tesla K 40. Z výsledkov vidíme, že program bežal najrýchlejšie pre GPU Tesla K40 a naopak najpomalšie pre GeForce GTX 470. Rozdiel medzi týmtito dvomi grafickými akcelerátormi nebol nijako závratný, približne 30 až 40 percent v prospech K 40.

Rozdiel medzi behom na CPU a GPU bol naopak priepastný - pre 24-vláknové CPU nám optimalizovaný algoritmus klasického násobenia bežal nad maticami o veľkosti

	256	**512**	**1024**	**2048**	**4096**	**8192**
GTX 470	0.207	1.455	11.270	89.944	717.77	5948.61
GTX 480	0.176	1.174	9.132	72.692	580.81	4849.28
GTX 590	0.173	1.258	9.846	78.502	627.67	5193.220
Tesla K40	0.140	0.936	7.509	59.764	476.84	3787.197

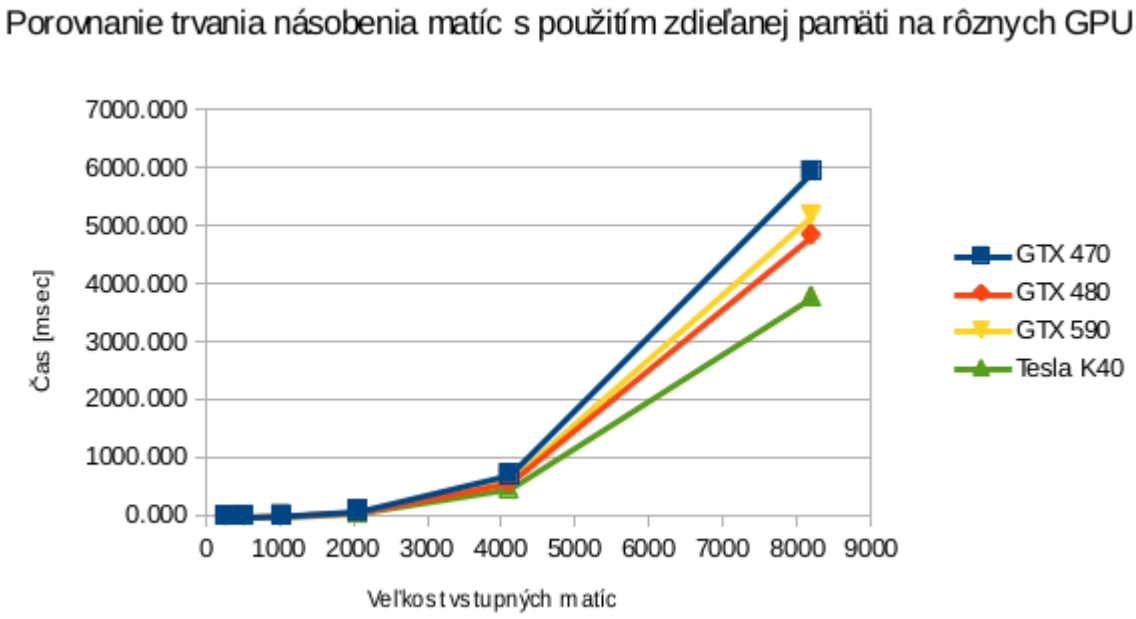
Table 2: Trvanie behu v ms v prostredí CUDA

4096 približne minútu, zatiaľ čo na GPU bežal nad rovnakým vstupom približne pol sekundy v závislosti od konkrétnej architektúry.

8 Záver

Podarilo sa nám implementovať a následne zparalelizovať ako CPU, tak aj GPU verziu násobenia matíc. Program pre CPU sme implementovali v 3 variantách - klasické násobenie, klasické násobenie s loop tilingom a Strassenov algoritmus. Pri následnej paralelizácii v prostredí OpenMP sme dosiahli lineárne zrýchlenie pre všetky tri implementácie. V prostredí GPU sme implementovali výhradne klasický algoritmus, Strassenov algoritmus sme z dôvodu problematickej implementácie rekurzie na GPU vynechali.

Prostredie GPU je mohutne paralelné, pri porovnaní rovnakých vstupných dát (matica 4096), klasický algoritmus s použitím loop tilingu, sme dosiahli 1000-násobné zrýchlenie (10 minút, presne 651.67s na 1-jadrovom CPU vs. 476.84ms na najrýchlejšom GPU, Tesla K40). Na druhej strane, programy v CUDA sú pomerne náročné na implementáciu, zatiaľ čo prostredie OpenMP je svojím high level prístupom veľmi príjemné pre programátora.


Porovnanie trvania násobenia matíc s použitím zdieľanej pamäti na rôznych GPU
7000.000
6000.000
5000.000
4000.000
3000.000
2000.000
1000.000
0.000
Čas [msec]
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Veľkosť vstupných matíc
GTX 470
GTX 480
GTX 590
Tesla K40